No.	Topics Covered
1	• Introduction to data structure. • Benefits of data structures. • Types of data structures. • How to select the suitable data structure. • Representation element in one dimensional array
2	• Representation element in two-dimensional array. • Representation element in array with structures.
3	• Stack: definition, operations, and algorithms
4	• Array representation of stack • Record implementation of stack
5	• Queue: definition, operations, and algorithms
6	• Array representation of Queue
7	• Record implementation of Queue
8	• Circular queue: definition, operations, and algorithms
9	• Array representation of Circular Queue
10	• Record implementation of Circular Queue
11	• Linked structure: sequential & dynamic Storage Allocation
12	• Linked list: definition, operations, and algorithms
13	• Linked Stack &Queue
14	• Double linked list
15	• Graphs: Directed graphs, Undirected graphs

16	• trees: Types of trees and its algorithms
17	• Transfer binary tree to ordinary tree & vice versa
18	• Transfer mathematical expression to binary tree & vise versa
19	• Tree representation
20	• Searching algorithms: sequential & binary search
21-27	• Sorting algorithms: bubble, insertion, quick, and hashing sorting

DATA SRUCTURE

USING C++ LANGUAGE

FOR COMPUTER SCIENCE STUDENTS / SECOND SATGE

LEC. : FAAZA ABDUL JABAR

Topic Covered

1. Introduction to data structure
2. Benefits of data structures
3. Types of data structures
4. How to select the suitable data structure
5. Representation element in one dimensional array
6. Representation element in array with structures
7. Stack: definition, operations and algorithms
8. Array representation of stack
9. Record implementation o stack
10. Queue: definition, operations and algorithms
11. Array representation of queue
12. Record implementation of queue
13. Circular Queue
14. Array representation of circular queue
15. Record implementation of circular queue
16. Linked structures: sequential and dynamic storage allocation
17. Linked list
18. Linked stack and queue
19. Double linked list
20. Graphs: directed and undirected
21. Trees: types and algorithms
22. Transfer binary tree to ordinary tree
23. Tree representation
24. Searching algorithms: sequential and binary search
25. Sorting algorithms: bubble, insertion, quick and hashing sorting

DATA SRUCTURE

USING C++ LANGUAGE

FOR COMPUTER SCIENCE STUDENTS / SECOND SATGE

LEC. : FAAZA ABDUL JABAR

1.1 Data Structures - Overview

Data Structure is a systematic way to organize data in order to use it efficiently. Following terms are the foundation terms of a data structure.

- **Interface** − Each data structure has an interface. Interface represents the set of operations that a data structure supports. An interface only provides the list of supported operations, type of parameters they can accept and return type of these operations.
- **Implementation** − Implementation provides the internal representation of a data structure. Implementation also provides the definition of the algorithms used in the operations of the data structure.

1.1.1 Characteristics of a Data Structure

- **Correctness** − Data structure implementation should implement its interface correctly.
- **Time Complexity** − Running time or the execution time of operations of data structure must be as small as possible.
- **Space Complexity** − Memory usage of a data structure operation should be as little as possible.

1.1.2 Need for Data Structure

As applications are getting complex and data rich, there are three common problems that applications face now-a-days.

- **Data Search** − Consider an inventory of 1 million(10^6) items of a store. If the application is to search an item, it has to search an item in 1 million(10^6) items every time slowing down the search. As data grows, search will become slower.
- **Processor speed** − Processor speed although being very high, falls limited if the data grows to billion records.
- **Multiple requests** − As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

To solve the above-mentioned problems, data structures come to rescue. Data can be organized in a data structure in such a way that all items may not be required to be searched, and the required data can be searched almost instantly.

1.2 Basic Concepts

This chapter explains the basic terms related to data structure.

1.2.1 Data Definition

Data Definition defines a particular data with the following characteristics.

- **Atomic** − Definition should define a single concept.
- **Traceable** − Definition should be able to be mapped to some data element.
- **Accurate** − Definition should be unambiguous.
- **Clear and Concise** − Definition should be understandable.

1.2.2 Data Object

Data Object represents an object having a data.

1.2.3 Data Type

Built-in Data Type

Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provide the following built-in data types.

- Integers
- Boolean (true, false)
- Floating (Decimal numbers)
- Character and Strings

Derived Data Type

Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. These data types are normally built by the combination of primary or built-in data types and associated operations on them. For example −

- List
- Array
- Stack
- Queue

1.3 Basic Operations

The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.

- Traversing
- Searching
- Insertion
- Deletion
- Sorting
- Merging

1.4 Abstract Data Types

An abstract data type (ADT) is an organized collection of information containing a set of operations used to manage that information. The set of operations includes methods such as add, delete, find etc. The set of operations/methods define the interface to the ADT

1.5 Algorithms

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms −

- **Search** − Algorithm to search an item in a data structure.
- **Sort** − Algorithm to sort items in a certain order.
- **Insert** − Algorithm to insert item in a data structure.
- **Update** − Algorithm to update an existing item in a data structure.
- **Delete** − Algorithm to delete an existing item from a data structure.

1.5.1 Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps or phases, and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

1.5.2 How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops do, for, while, flow-control if–else, etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

Example

Let's try to learn algorithm-writing by using an example.

Problem − Design an algorithm to add two numbers and display the result.

Step 1 − START
Step 2 − declare three integers **a**, **b** & **c**
Step 3 − define values of **a** & **b**
Step 4 − add values of **a** & **b**
Step 5 − store output of **step 4** to **c**
Step 6 − print **c**
Step 7 − STOP

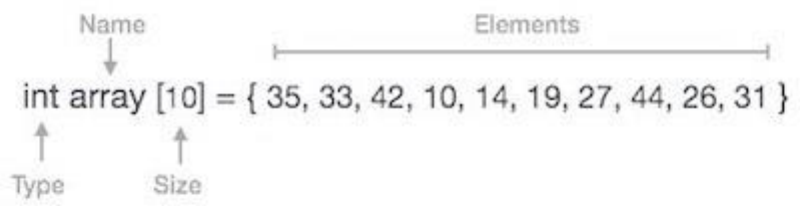
2.1 Array

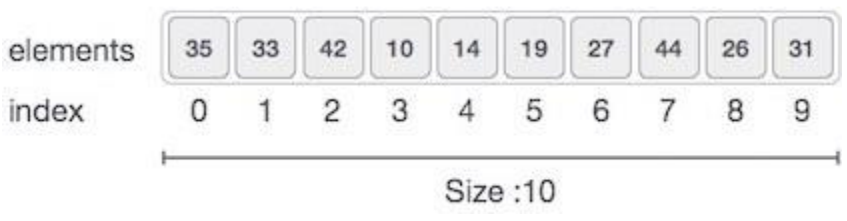
Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

- **Element** − Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

2.2 Array Representation

Arrays can be declared in various ways in different languages. For illustration, let's take C++ array declaration.





As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index.

2.3 Basic Operations

Following are the basic operations supported by an array.

- **Traverse** − print all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.

2.3.1 Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.
Here, we see a practical implementation of insertion operation, where we add data at the end of the array −

Algorithm

Let **Array** be a linear unordered array of **MAX** elements.

Let **LA** be a Linear Array unordered with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm where ITEM is inserted into the K^{th} position of LA –

```
1. Start
2. Set J = N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop
```

Following is the implementation of the above algorithm –

```
#include <stdio.h>
main() {
   int LA[] = {1,3,5,7,8};
   int item = 10, k = 3, n = 5;
   int i = 0, j = n;
   cout<<"The original array elements are :\n";

   for(i = 0; i<n; i++) {
         cout<<"LA["<<i<<"] ="<< LA[i]<<"\n";
   }

   n = n + 1;

   while( j >= k) {
      LA[j+1] = LA[j];
      j = j - 1;
   }

   LA[k] = item;

   cout<<"The array elements after insertion :\n";

   for(i = 0; i<n; i++) {
         cout<<"LA["<<i<<"] ="<< LA[i]<<"\n";
   }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after insertion :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 10
LA[4] = 7
LA[5] = 8
```

2.3.2 Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to delete an element available at the Kth position of LA.

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

Example

Following is the implementation of the above algorithm –

```
#include <stdio.h>
void main() {

  int LA[] = {1,3,5,7,8};
  int k = 3, n = 5;
  int i, j;

   cout<<"The original array elements are :\n";

   for(i = 0; i<n; i++) {
      cout<<"LA["<<i<<"] ="<< LA[i]<<"\n";
   }

   j = k;
   while( j < n) {
      LA[j-1] = LA[j];
      j = j + 1;
   }
   n = n -1;
      cout<<"The array elements after deletion :\n";
   for(i = 0; i<n; i++) {
         cout<<"LA["<<i<<"] ="<< LA[i]<<"\n";
   }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after deletion :
LA[0] = 1
LA[1] = 3
LA[2] = 7
LA[3] = 8
```

2.3.3 Search Operation

You can perform a search for an array element based on its value or its index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Example

Following is the implementation of the above algorithm –

```
#include <stdio.h>

void main() {
   int LA[] = {1,3,5,7,8};
   int item = 5, n = 5;
   int i = 0, j = 0;
      cout<<"The original array elements are :\n";
```

```
 for(i = 0; i<n; i++) {
     cout<<"LA["<<i<<"] ="<< LA[i]<<"\n";

  }

  while( j < n){
     if( LA[j] == item ) {
        break;
     }

     j = j + 1;
  }

  Cout<<"Found element "<<item<<"at position"<<j+1<<"\n";
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :

LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
Found element 5 at position 3
```

2.3.4 Update Operation

Update operation refers to updating an existing element from the array at a given index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to update an element available at the Kth position of LA.

```
1. Start

2. Set LA[K-1] = ITEM

3. Stop
```

Example

Following is the implementation of the above algorithm –

```
#include <stdio.h>
void main() {
   int LA[] = {1,3,5,7,8};
   int k = 3, n = 5, item = 10;
   int i, j;

   cout<<"The original array elements are :\n";

   for(i = 0; i<n; i++) {
     cout<<"LA["<<i<<"] ="<< LA[i]<<"\n";
   }

   LA[k-1] = item;

   Cout<<"The array elements after updation :\n";

   for(i = 0; i<n; i++) {
     cout<<"LA["<<i<<"] ="<< LA[i]<<"\n";   }
}
```

When we compile and execute the above program, it produces the following result −

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after updation :
LA[0] = 1
LA[1] = 3
LA[2] = 10
LA[3] = 7
LA[4] = 8
```

3.1 Stack

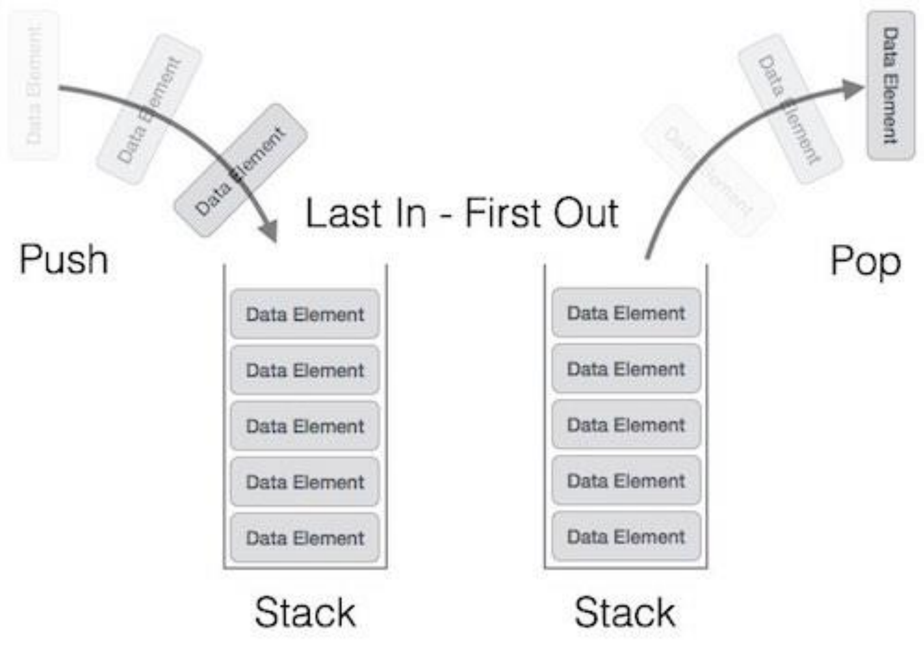
A stack is an Abstract Data Type ADT, commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack **ADT** allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it **LIFO** data structure. **LIFO** stands for Last-in-first-out. Here, the element which is placed inserted or added last, is accessed first. In stack terminology, insertion operation is called PUSH operation and removal operation is called **POP** operation.

3.1.1 Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

3.1.2 Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

- **Push ( )**− Pushing storing an element on the stack.
- **Pop ( )** − Removing accessing an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

- **Peek ()** − get the top data element of the stack, without removing it.
- **isFull** () − check if stack is full.
- **isEmpty** () − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

Peek ()

Algorithm of peek () function −

```
begin procedure peek
   return stack[top]
end procedure
```

Implementation of peek function in C programming language −

Example

```
int peek() {

   return stack[top];

}
```

Isfull ()

Algorithm of isfull () function –

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Implementation of isfull function in C programming language –

Example

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
}
```

Isempty () : Algorithm of isempty () function –

```
begin procedure isempty

   if top less than 1
      return true
   else
      return false
   endif

end procedure
```

Implementation of isempty function in C ++ programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

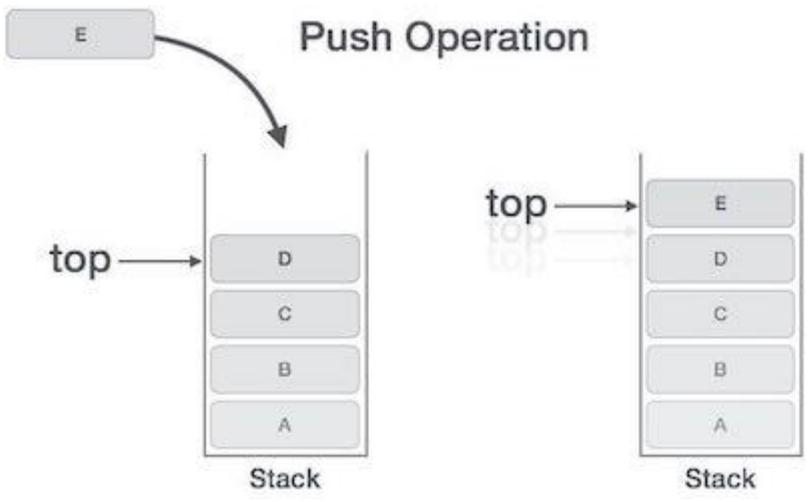
Example

```
bool isempty() {

   if(top == -1)

      return true;

   else

      return false;

}
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data
```

```
   if stack is full
      return null
   endif

   top ← top + 1
   stack[top] ← data

end procedure
```

Implementation of this algorithm in C++, is very easy. See the following code –

Example

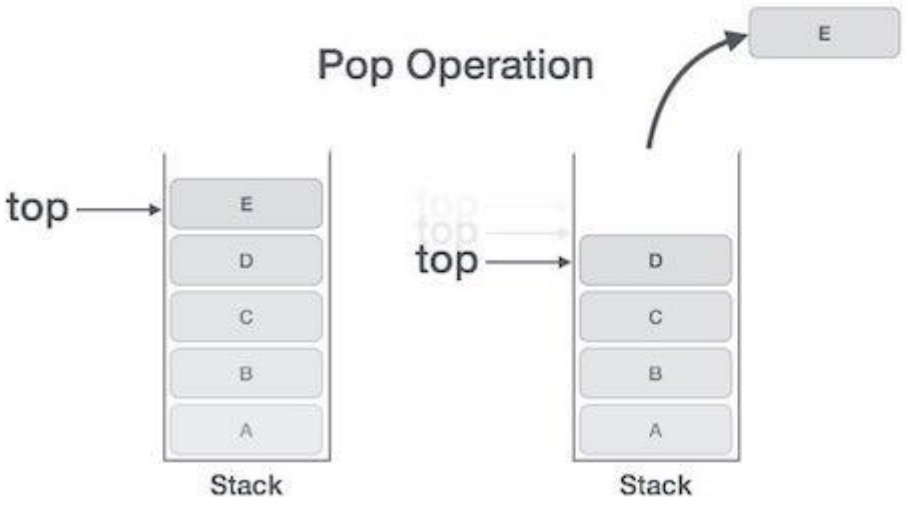
```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      Cout<<"Could not insert data, Stack is full.\n";
   }
}
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop actually removes data element and de-allocates memory space.

A Pop operation may involve the following steps –

- **Step 1** – Checks if the stack is empty.
- **Step 2** – If the stack is empty, produces an error and exit.
- **Step 3** – If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** – Decreases the value of top by 1.
- **Step 5** – Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows –

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]
   top ← top - 1
   return data

end procedure
```

Implementation of this algorithm in C++, is as follows –

Example

```
int pop(int data) {

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
        Cout<<"Could not retrieve data, Stack is empty.\n";
    }
}
```

3.2 Expression Parsing

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

- Infix Notation
- Prefix Polish Notation
- Postfix Reverse− Polish Reverse− Polish Notation

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

Infix Notation

We write expression in **infix** notation, e.g. a - b + c, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is **prefix**ed to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

Postfix Notation

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is postfixed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

The following table briefly tries to show the difference in all three notations −

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	a+b * c	* + a b c	a b + c *
3	a ∗ b+c	∗ a + b c	a b c + ∗
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	a+b ∗ c+d	∗ + a b + c d	a b + c d + ∗
6	(a+b ∗ c) - d	- ∗ + a b c d	a b + c ∗ d -

Parsing Expressions

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example −

$$a + b * c \Rightarrow a + (b * c)$$

As multiplication operation has precedence over addition, b * c will be evaluated first. A table of operator precedence is provided later.

Associativity

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a + b − c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as a+b − **c**.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table highest to lowest −

Sr.No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication ∗∗ & Division //	Second Highest	Left Associative
3	Addition ++ & Subtraction −−	Lowest	Left Associative

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example −

In **a + b*c**, the expression part **b***c** will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for **a + b** to be evaluated first, like a+b***c**.

Postfix Evaluation Algorithm

We shall now look at the algorithm on how to evaluate postfix notation −

```
Step 1 − scan the expression from left to right
Step 2 − if it is an operand push it to stack
Step 3 − if it is an operator pull operand from stack and perform operation
Step 4 − store the output of step 3, back to stack
Step 5 − scan the expression until all operands are consumed
Step 6 − pop the stack and perform operation
```

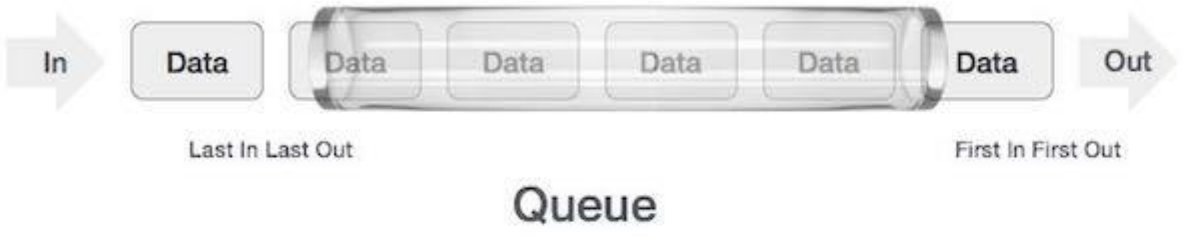
3.3 Queue

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data enqueue and the other is used to remove data dequeue. Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

3.3.1 Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −



As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

3.3.2 Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

- **enqueue** − add (store) an item to the queue.
- **dequeue** − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

- **peek** − Gets the element at the front of the queue without removing it.

- **isfull** − Checks if the queue is full.
- **isempty** − Checks if the queue is empty.

In queue, we always dequeue or access data, pointed by **front** pointer and while enqueing or storing data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue −

peek

This function helps to see the data at the **front** of the queue. The algorithm of peek function is as follows −

Algorithm

```
begin procedure peek
   return queue[front]
end procedure
```

Implementation of peek function in C programming language −

Example

```
int peek() {

   return queue[front];

}
```

isfull

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull function −

Algorithm

```
begin procedure isfull


   if rear equals to MAXSIZE

      return true

   else

      return false

   endif
```

```
end procedure
```

Implementation of isfull function in C programming language –

Example

```c
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

isempty

Algorithm of isempty function –

Algorithm

```
begin procedure isempty

   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif

end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code –

Example

```c
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
```

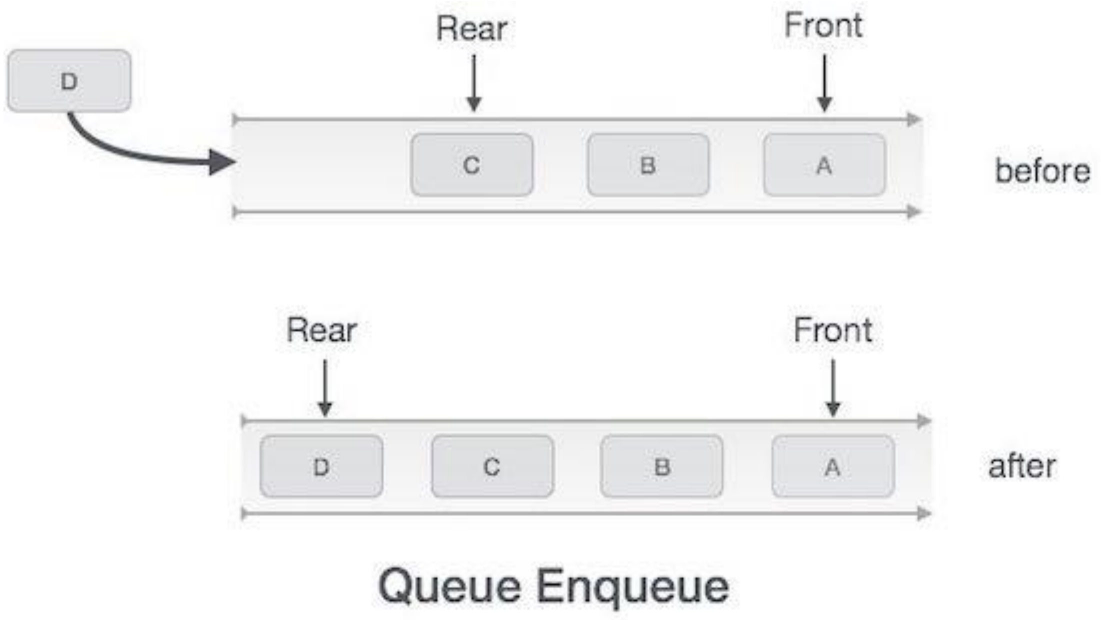
```
      return false;
}
```

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.



Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

Algorithm for enqueue operation

```
procedure enqueue(data)


   if queue is full
      return overflow
   endif
   rear ← rear + 1
   queue[rear] ← data
   return true

end procedure
```

Implementation of enqueue in C programming language −

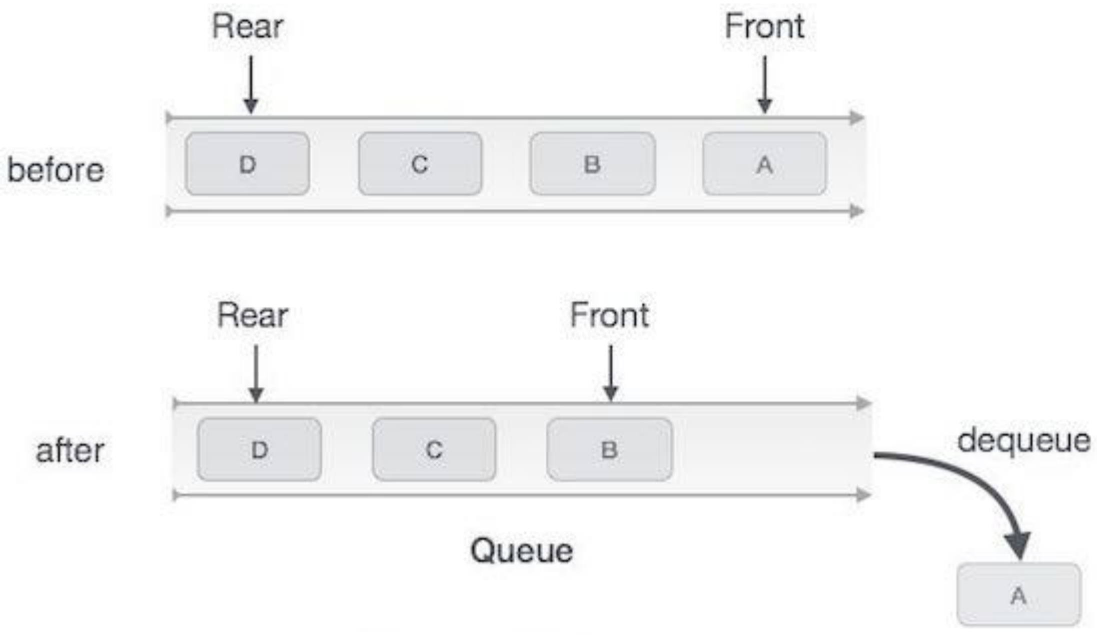
Example

```
int enqueue(int data)
   if(isfull())
      return 0;

   rear = rear + 1;
   queue[rear] = data;

   return 1;
end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue

   if queue is empty
      return underflow
   end if

   data = queue[front]
   front ← front + 1
   return true

end procedure
```

Implementation of dequeue in C programming language –

Example

```
int dequeue() {
   if(isempty())
      return 0;

   int data = queue[front];
   front = front + 1;


   return data;
}
```